UC Berkeley – Computer Science
CS61BL: Data Structures
Midterm 1, Summer 2017

This test has 8 questions worth a total of 30 points, and is to be completed in 110 minutes. The exam is closed book, except that you are allowed to use one double-sided page of notes as a cheat sheet (front and back). No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided.

Write the statement out below in the blank provided and sign. You may do this before the exam begins. **Any plagiarism, no matter how minor, will result in points deducted from your exam.**

"I have neither given nor received any assistance during the taking of this exam."

_____

_____

Signature: _____

Write your name and student ID on the front page. Write the names of your neighbors. Write and sign the above statement. Once the exam has started, write your class ID in the corner of every page.

Name: _____ **Your Class ID: _____**

SID: _____ **Name of person to left: _____**

TA: _____ **Name of person to right: _____**

Tips:

- There may be partial credit for incomplete answers. Write as much of the solution as you can, but bear in mind that we may deduct points if your answers are much more complicated than necessary.
- There are a lot of problems on this exam. **Work through the ones with which you are comfortable first. Do not get overly captivated by interesting design issues or complex corner cases you're not sure about.**
- Not all information provided in a problem may be useful.
- Unless otherwise stated, all given code on this exam should compile. All code has been compiled and executed before printing, but in the unlikely event that we do happen to catch any bugs during the exam, we'll announce a fix. Unless we specifically give you the option, the correct answer is not 'does not compile.'
- ○ indicates that only one circle should be filled in.
- ☐ indicates that more than one box may be filled in.
- For answers which involve filling in a ○ or ☐, **please fill in the shape completely.**

Optional. Mark along the line to show your feelings on the spectrum between ☹ and ☺.

Before exam: [☹_____☺].

After exam: [☹_____☺].

1. Dive to the Heart (4 pts)

Write the full output of attempting to compile and run the following programs. In the case of **compiler** errors, write "*Compiler Error*". In the case of **runtime** exceptions, you should write the output of lines that execute before the exception occurs in addition to "*Runtime Exception*". Each `main` method is run independently of the others.

Code	Output

`public class Palette {`
`  String color;`
 	
`  public Palette blackOut() {`
`  color = "black";`
`  return this;`
`  }`
 	
`  public static void main(String[] args) {`
`  Palette iu = new Palette();`
`  iu.color = "red";`
`  Palette gd = iu.blackOut();`
`  System.out.println(gd.color);`
`  gd.color = "peach";`
`  System.out.println(gd.color);`
`  System.out.println(iu.color);`
`  }`
`}`	

`public class Mym {`
`  String xaler;`
 	
`  public Mym(String s) {`
`  this.xaler = s;`
`  }`
 	
`  public static void eterize(Object[] arr) {`
`  for (int i = 0; i < arr.length; i++) {`
`  Mym lay = (Mym) arr[i];`
`  System.out.println(lay.xaler);`
`  }`
`  }`
 	
`  public static void main(String[] args) {`
`  Object[] arr = {new Mym("cs61bl"),`
`  new Mym("rox"), "covfefe"};`
`  eterize(arr);`
`  }`
`}`	

Code	Output

`public class Astro {`
`  public static int dreams = 0;`
 	
`  public void realize(int time) {`
`  time = dreams + time;`
`  dreams = time;`
`  System.out.println(dreams);`
`  }`
 	
`  public static void main(String[] args) {`
`  Astro rocky = new Astro();`
`  rocky.realize(10);`
`  Astro moon = new Astro();`
`  moon.realize(10);`
`  System.out.println(Astro.dreams);`
`  }`
`}`	

`public class CherryBomb {`
 	
`  int leaves = 0;`
 	
`  public void explode(String s) {`
`  s = "BOOM!";`
`  leaves += 1;`
`  System.out.println(s);`
`  }`
 	
`  public static void main(String[] args) {`
`  CherryBomb c = new CherryBomb();`
`  c.leaves = 10;`
`  String s = "Whiplash";`
`  c.explode(s);`
`  System.out.println(s);`
`  System.out.println(c.leaves);`
`  }`
`}`	

2. Enchanted Dominion (4 pts)

Consider the following implementation of `IntList`:

```
public class IntList {
    public int head;
    public IntList tail;
    public IntList(int head, IntList tail) {
        this.head = head; this.tail = tail;
    }

    public static void swapTails(IntList first, IntList second) {
        IntList temp = first.tail;
        first.tail = second.tail;
        second.tail = temp;
    }

    public static void swapHeads(int x, int y) {
        int temp = x;
        x = y;
        y = temp;
    }
}
```

a. The following code is executed **in order**. There are no compiler or runtime errors. For each print statement, fill in the bubble **completely** corresponding to the integer that's printed. You will be asked to draw two box-and-pointer diagrams on the next page.

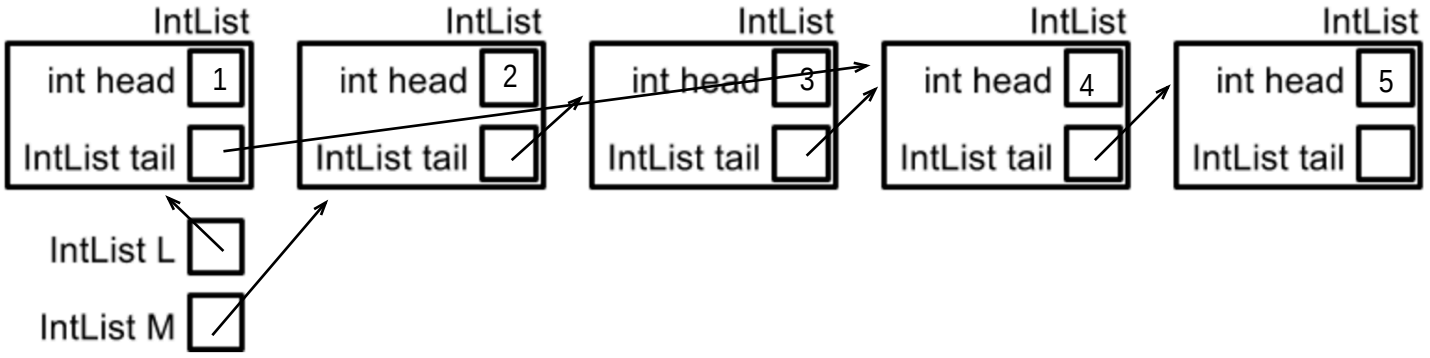
Statement(s)	1	2	3	4	5	6
`IntList L = new IntList(1, new IntList(2, new IntList(3, new IntList(4, null))));` `System.out.println(L.tail.tail.head);`	○	○	●	○	○	○

`IntList M = L.tail.tail;`
`System.out.println(M.tail.head);`
`// Draw the state of the program on next page`
	○	○	○	●	○	○

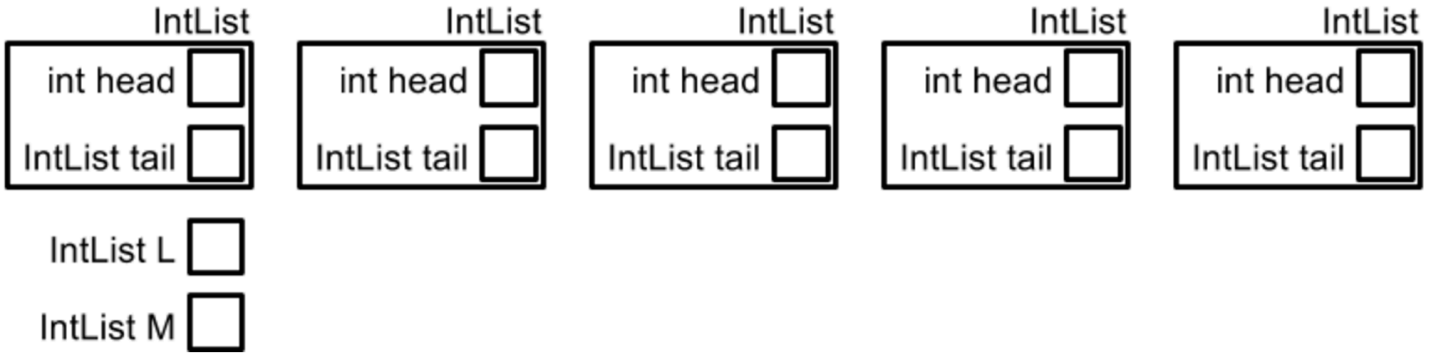
`M.tail.tail = new IntList(5, null);`
`IntList.swapTails(L, M);`
`System.out.println(M.tail.head);`
	○	○	●	○	○	○

`M.tail.tail.head = 6;`
`IntList.swapHeads(M.head, L.head);`
`System.out.println(M.head);`
	○	●	○	○	○	○

b. Now, fill in the box-and-pointer diagram below to represent the state of the program after the execution of **the 2nd box** above. Not all boxes may be needed.



c. Now, fill in the box-and-pointer diagram below to represent the state of the program after the execution of the **final line** above. Not all boxes may be needed.



3. Symphony of Sorcery (2 pts)

Consider the two classes defined below:

```
public class Patriot {
    public String name;
    public Patriot(String name) {
        this.name = name;
    }
    public void politicate() {
        System.out.println("Give me liberty or give me death!");
    }
}

public class Federalist extends Patriot {
    public Federalist(String name) {
        super(name);
    }
    public void politicate() {
        System.out.println("Defend the union!");
    }
}
```

You now execute the below lines of code, **in order**. If the line(s) in a given box result in an error, mark the bubble corresponding to the appropriate error (compiler or runtime error). Otherwise, mark the bubble corresponding to the appropriate outputted value (Ø, P, or F from the "Print Output Table"). Fill in all bubbles **completely**. The first two boxes are done for you.

Print Output Table

Code	Output
Ø	No print output.
P	Give me liberty or give me death!
F	Defend the union!

Line(s)	**Compiler Error?**	**Runtime Error?**	**Ø**	**P**	**F**
`Patriot wash = new Patriot("Washington");`	○	○	●	○	○
`Patriot ham = new Federalist("Hamilton");`	○	○	●	○	○
`ham.politicate();`	○	○	○	○	○
`Federalist jay = new Patriot("Jay");` `jay.politicate();`	○	○	○	○	○
`Federalist ham2 = ham;`	○	○	○	○	○
`Federalist wash2 = (Federalist) wash;`	○	○	○	○	○
`Federalist ham3 = (Federalist) ham;` `((Patriot) ham3).politicate();`	○	○	○	○	○

4. Mysterious Tower (4 pts)

a. Fill in the square next to the lines of code that can **never** cause a runtime error (not including system errors such as `StackOverflowError`, `OutOfMemoryError`, etc.). You may assume that **all code compiles and method bodies do not contain code that would cause a runtime error.**

☐ `if (xianth.blurg == 10)`

☐ `Zeeg zarg = new Zeeg();`

☐ `while (yebi != null && yebi.mianate())`

☐ `int zel = da[10];`

☐ `Dylth id = (Dylth) xeonite;`

☐ `Charp vir = gion;`

b. For each of the following propositions, fill in true or false **completely**. If the proposition is false, provide a counterexample $f(n)$ and $g(n)$. Assume $f(n)$ and $g(n)$ are positive, strictly increasing functions.

○ True / ○ False: If $f(n) \in \Omega(g(n))$, then $g(n)$ **must be** in $O(f(n))$

$f(n)$: _____ $g(n)$: _____

○ True / ○ False: If $f(n) \in O(g(n))$, then $g(n)$ **cannot be** in $O(f(n))$

$f(n)$: _____ $g(n)$: _____

○ True / ○ False: If $f(n) \in O(2^{g(n)})$ and $f(n) \in \Omega(1)$, then $g(n)$ **must be** in $\Theta(1)$

$f(n)$: _____ $g(n)$: _____

○ True / ○ False: If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n)$ **must be** in $O(h(n))$

$f(n)$: _____ $g(n)$: _____

○ True / ○ False: If $f(n) \in O(g(n))$, then $\lg f(n)$ **must be** in $O(\lg g(n))$

$f(n)$: _____ $g(n)$: _____

○ True / ○ False: If $\lg f(n) \in O(\lg g(n))$, then $f(n)$ **must be** in $O(g(n))$

$f(n)$: _____ $g(n)$: _____

5. Daybreak Town (3 pts)

a. Consider the `SLList` class, which represents a singly-linked list. A heavily abridged version of this class appears below:

```
public class SLList {
    ...
    /* Construct an empty SLList. */
    public SLList() { ... }
    /* Adds x to the front of the list. */
    public void insertFront(int x) { ... }
    /* Returns the index of x in the list, if it exists.
       Otherwise, returns -1. */
    public int indexOf(int x) { ... }
}
```

"Well that's dumb," you observe after reading `indexOf`'s comment, "-1 isn't a real index. That method should produce an error in that case instead." You decide to write the `SLListVista` class, which must have all of the functionality of `SLList`, except **`SLListVista`'s `indexOf` method produces a `NoSuchElementException` in the event that x isn't present in the list.** `NoSuchElementException` is an unchecked exception. In the space provided, fill in the `SLListVista` class. You may not need all lines. Each line should contain only one statement.

```
import java.util.NoSuchElementException;

public class SLListVista ____________________________________ {

     @Override

     ____________________________________________________________

     ____________________________________________________________

     ____________________________________________________________

     ____________________________________________________________

     ____________________________________________________________

     ____________________________________________________________

     ____________________________________________________________

     ____________________________________________________________

}
```

b. Now that you've written your rad `SLListVista` class, you must test it! Fill in the test below to confirm that `indexOf` correctly throws a `NoSuchElementException` **when called on an empty list**. Your test should pass **if and only if** a `NoSuchElementException` is thrown by `indexOf`. You may assume `SLListVista`, `NoSuchElementException`, and JUnit are already imported. You may not need all lines. Each line should contain only one statement. You might find the `assertTrue(boolean condition)` method helpful.

```
@Test
public void testIndexOfEmpty() {
    _____________________________________________________
    _____________________________________________________
    _____________________________________________________
    _____________________________________________________
    _____________________________________________________
    _____________________________________________________
    _____________________________________________________
    _____________________________________________________
    _____________________________________________________
    _____________________________________________________
}
```

6. Radiant Garden (5 pts)

Complete the `expand` method in the `DLList` class, which mutates a circular doubly-linked list with a sentinel node and `expand`s it such that it passes the JUnit tests below. When `expand` is called on the list `(1 3 5)`, the list is mutated to `(1 3 3 3 5 5 5 5 5).` That is, the original value of `list.get(0)` gets repeated `list.get(0)` times in the output list. This is then followed by the original value of `list.get(1)` repeated `list.get(1)` times, et cetera. Your solution should modify the `DLList` instance that it is called on without constructing any new `DLList`s. Assume all input list elements are larger than 0. For full credit, the `DLList` must be well formed (all pointers are correct). Each line should contain only one statement.

```
import static org.junit.Assert.*;
import org.junit.Test;

public class DLList {

    private DLNode sentinel;

    public DLList() { ... }

    private class DLNode {
        private int item;
        private DLNode prev, next;

        public DLNode(int i, DLNode p, DLNode n) {
            item = i;
            prev = p;
            next = n;
        }
    }

    @Override
    public boolean equals(Object o) { ... }
    public static DLList list(int... args) { ... }
    public int get(int index) { ... }

    @Test
    public void testExpand() {
        DLList d = DLList.list(1, 2, 3);
        d.expand();
        assertEquals(DLList.list(1, 2, 2, 3, 3, 3), d);
        DLList d2 = DLList.list(2, 1);
        d2.expand();
        assertEquals(DLList.list(2, 2, 1), d2);
        d2.expand();
        assertEquals(DLList.list(2, 2, 2, 2, 1), d2);
    }
```

```
public void expand() {
    Dlist refer = sentinel.next                                    ;
    while (  refer.next != sentinel                                ) {
        ____________________________________________________________;
        for (________________;________________;________________) {
            ____________________________________________________;
            ____________________________________________________;
            ____________________________________________________;
            ____________________________________________________;
        }
        ____________________________________________________;
    }
}
}
```

0. PNH (0 pts)

Hemolytic anemia is a class of anemia caused by premature breakdown of red blood cells in the body. There is only one form of hemolytic anemia caused by an acquired intrinsic defect in the red blood cell membrane. What is its name?

Newsflash: Wild Jarmigon Spotted

7. The Grid (4 pts)

Implement the `gridMatch` method, which returns true if `sub` is a subgrid of `matrix`. Otherwise, it returns false. `sub` is a subgrid if it matches a contiguous section of the input `matrix`. You may assume both `sub` and `matrix` are rectangular. You may assume that neither `matrix` nor `sub` have any dimensions of zero.

Examples:

Input Matrix	Input Sub	Matching portion	Return Value
1 0 3 4 5 3 2 1 7 1 2 0 8 0 3 0	3 2 1 2	1 0 3 4 5 **3** **2** 1 7 **1** **2** 0 8 0 3 0	true
1 0 3 4 -5 3 2 1 7 1 -2 0 8 0 3 0	-5 3 2 7 1 -2	1 0 3 4 **-5** **3** **2** 1 **7** **1** **-2** 0 8 0 3 0	true
1 0 3 4 5 3 2 1 7 1 2 0 8 0 3 0	1 2 1 2	None	false

```
public static boolean gridMatch(int[][] matrix, int[][] sub) {
    boolean contains = false;
    for (int i = 0; i < _____; i++) {
        for (int j = 0; j < _____; j++) {
            _____;
            for (int a = 0; a < _____; a++) {
                for (int b = 0; b < _____; b++) {
                    _____;
                }
            }
            _____;
        }
    }
    return false;
}
```

8. Mirage Arena (4 pts)

`Gitlit` is a version control system that records successive versions of a `String`. Newly constructed `Gitlit`s have a default initial backup: "Initial version." Fill in the `Gitlit` class so that the JUnit tests on the next page pass. For full credit, your `Gitlit` solution **must** behave according to its comments. You may not need all lines. Each line should contain only one statement.

```
import static org.junit.Assert.*;
import org.junit.Test;

public class Gitlit {
    public Commit head = ____________________________________;
    public static final String ERR_MSG = "The requested commit doesn't exist.";

    /** Add a backup to Gitlit. */
    public void recordBackup(String backup) {
        head = new Commit(backup, head);
    }

    /** Get the backup i backups ago. If i is zero, this method returns the most
        recent backup. If i is one, this method returns the second most recent
        backup, etc. If i is invalid (e.g. if it's < 0 or if it's >= the number
        of backups), returns Gitlit.ERR_MSG. */
    public String getBackup(int i) {
        if (i < 0) return Gitlit.ERR_MSG;
        return head.getBackup(i);
    }

    public class Commit {
        public String backup;
        public Commit tail;

        public Commit(String backup, Commit tail) {
            this.backup = backup; this.tail = tail;
        }

        public String getBackup(int i) {
            if (i == 0) {
                return backup;
            } else {
                return tail.getBackup(i - 1);
            }
        }
    }
```

... (continued on next page)

```
    public class ____________________________________________ {

        ____________________________________________________
        ____________________________________________________
        ____________________________________________________
        ____________________________________________________
        ____________________________________________________
        ____________________________________________________
        ____________________________________________________
        ____________________________________________________
        ____________________________________________________
        ____________________________________________________

    }

    @Test
    public void testGetBackup() {
        Gitlit g = new Gitlit();
        assertEquals("Initial version.", g.getBackup(0));
        g.recordBackup("Update 1.");
        g.recordBackup("Update 2.");
        assertEquals("Update 1.", g.getBackup(1));
        assertEquals("Initial version.", g.getBackup(2));
        assertEquals(Gitlit.ERR_MSG, g.getBackup(3));
        assertEquals(Gitlit.ERR_MSG, g.getBackup(4));
    }
}
```